SDVS 1994 Final Report

30 September 1994

Prepared by

L. G. MARCUS
Trusted Computer Systems Department
Computer Science and Technology Subdivision
Computer Systems Division
Engineering and Technology Group

19971007 084

Prepared for

DEPARTMENT OF DEFENSE
Ft. George G. Meade, MD 20744-6000

DTIC QUALITY INSPECTED 4

Engineering and Technology Group

THE AEROSPACE CORPORATION
El Segundo, California

SDVS 1994 FINAL REPORT

Prepared by

L. G. MARCUS
Trusted Computer Systems Department
Computer Science and Technology Subdivision
Computer Systems Division
Engineering and Technology Group

30 September 1994

Engineering and Technology Group
THE AEROSPACE CORPORATION
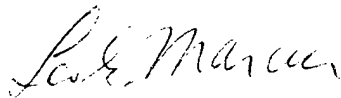El Segundo, CA 90245-4691

Prepared for

DEPARTMENT OF DEFENSE
Ft. George G. Meade, MD 20744-6000

SDVS 1994 FINAL REPORT

Prepared by

L. G. Marcus

Approved by

D. B. Baker, Director
Trusted Computer Systems Department

Abstract

This report details the progress made on the SDVS project's FY 94 tasks funded by the United States Department of Defense through Air Force Space and Missile Systems Center contract number F04701-93-C-0094.

Acknowledgments

Contents

1 Introduction

The purpose of the State Delta Verification System (SDVS) project is to address some of the inadequacies of the currently popular certification and analysis methods (e.g. testing, simulation, and design walkthroughs) for assuring the correctness of computer systems.

This ongoing effort has primarily focused on developing a theoretical framework and software tools for the formal verification of computer systems. SDVS is an automated system to help write and check proofs of the correctness of computer systems with respect to formal specifications ([1], [2] or [3]).

The current version of SDVS (SDVS 13) includes capabilities in three main areas:

1. High level software (Ada) verification; see [4].

2. Microcode level (ISPS) verification; see [5].

3. Hardware (VHDL) verification; see [6].

SDVS 13 differs from SDVS 12 primarily in its enhanced VHDL verification capabilities. SDVS 13 runs on the Sparc architecture.

The main focus of the body of this report will be on the progress of this year's tasks that were funded by the United States Department of Defense through Air Force Space and Missile Systems Center contract number F04701-93-C-0094.

We begin this report with brief background material about SDVS in Section 2, in which we give information about the three main modules of SDVS: the proof environment, the translators, and the user interface.

The remaining sections in this report discuss progress in Fiscal Year 1994. In particular, Section 3 contains a discussion of a significant application of SDVS, the verification of a real DoD Ada program, and SDVS enhancements implemented for this application. Section 4 discusses the adaptation and demonstrations of SDVS to handle increasingly complex subsets of VHDL, including progress on the formal verification of a real NSA application.

General system development, upgrades to the user documention, and user support are summarized in Section 5. Section 6 reports on some of the work done under this contract on formal specification languages. The conclusions are presented in Section 7.

Aerospace distributes all versions of SDVS, although the distribution must have prior approval by the government sponsor. The current distribution policy is that SDVS can be distributed to universities and companies in the United States. It is still under export control, and organizations requesting SDVS outside the United States must use a separate request procedure.

The SDVS project is one aspect of a larger push to bring formal methods for assurance of computer system correctness to the level where their potential benefits can be realized on real computer systems applications ("technology transition").

Thus, in addition to SDVS, we have recently turned to the so-called industrial strength formal specification languages (or FDTs: formal description techniques, such as Z, LOTOS, or VDM), both as assurance tools in their own right, and also for their potential use in formal verification and testing.

A parallel effort, though not funded under this contract, involves exploring the possibilities of obtaining rigorous probabilistic results when a full proof is not feasible [7].

Technology transition has seen some isolated successes, primarily in the field of the formal specification languages mentioned above, where it is generally agreed that the pay-off is more immediate, more easily achieved, and more beneficial to the design process. Looking to the near future, it is natural to expect that verification systems will be able to prove the correspondence between a specification written in an FDT and a system written in some standard programming language. Currently, as mentioned above, SDVS accepts programs written in standard programming languages *verbatim* and performs the translation into logic in a traceable manner within the system operation. Extending this translation capability to an FDT is a project we hope to undertake at some point in the near future. Another effort is reported on in [8].

The current SDVS effort has yielded several lessons.

The accepted wisdom is that verification is difficult, if not impossible. This statement has sociological and technical aspects. Certainly someone about to undertake a verification project will probably have to learn some new skills, depending on his or her previous knowledge, but it is reasonable to assume that there is nothing superhuman about the ability of logicians, mathematicians, or developers of verification systems that would preclude a motivated software engineer from acquiring the necessary skills. Of course, the final verdict should be based on scientific sociological studies of this phenomenon.

Two dimensions of any verification system are what the system can do and how easy it is to do that. Each of these dimensions naturally breaks down into many components. The study of the interrelationships of these components with respect to the success of technology transition is another interesting and important topic which has not been addressed, as far as we know.

The Ada MSX verification experiment reported on here was in fact very difficult. It was difficult to write the specifications in a formal logical manner (in the state delta language) and it was difficult to carry out the proof. An added difficulty was that SDVS was being developed as the proof was being carried out. It is likely that a similar verification exercise could be carried out on the new SDVS much more easily because of the new capabilities of SDVS, and if the same people were to do the new verification, their experience would make a similar job much simpler the second time around.

Another lesson is that simply the act of preparing the patient for the operation (the computer system for formal analysis) can yield important benefits, e.g. bug discoveries, as was the case for the MSX project. And at the end of the operation, something was in fact proved, thereby further increasing the confidence in the correctness of the system.

Two final thoughts: (1) Formal verification is most suited for critical (life, safety, or security-

critical) systems or modules of systems. Thus, "design for verification" must imply isolating the critical functions in their own modules. (2) A system implementor writing critical code should have a very clear idea about why the code conforms to the specification. If that idea is not formalizable and provable, then perhaps in fact the implementor's intuition is flawed.

2 SDVS Background

As mentioned above, the primary emphasis of this project has been the development of the State Delta Verification System. SDVS is one of a class of automated verification systems that have been developed over the last decade or so at various academic and industrial institutions across the US and abroad. The various systems embody differing approaches with regard to their theoretical foundations, their modes of operation, and their intended applications.

There are several features that characterize SDVS (a more detailed discussion is given in [2]). SDVS has a theorem prover/proof checker that can run in interactive or batch modes; the user supplies high-level proof commands, while many low-level proof deductions are executed automatically. One of the more distinctive features of SDVS is its potential to incorporate widely used application languages (e.g. subsets of VHDL, Ada, and ISPS) and use descriptions in the application languages as either implementations or, in some cases, specifications in the verification hierarchy. Translators are used to translate application languages automatically to the internal logical language of SDVS, the state delta logic. The state delta logic is equivalent to a part of classical temporal logic; it can express the "always," "eventually," and a version of the "until" operators [9, 10]. The "mod-list" feature and "covering" algorithm present a unique method for keeping track of facts that are preserved through state transitions. SDVS has a generous amount of built-in domain knowledge, including for example, integer and bitstring arithmetic. A solver (EKL [11]) for a fragment of full first order logic has been incorporated into the quantification deduction mechanism. The SDVS simplifier is being used in other verification systems, e.g. Penelope [12].

2.1 SDVS Proof

Proofs in SDVS are technically proofs of state delta formulas (or simply, "state deltas"). Without giving the full definition, we can say that a state delta is essentially a compact description of a conditional state transition, with "handles" for stating, among other things, which variables are held constant and which formulas are invariant over the transition. State deltas can be used to represent programs, hardware descriptions, specifications, and correctness theorems. The semantics of state deltas is defined with respect to "computational models;" in brief, a computational model is a sequence of states. A state delta is valid if and only if it holds in all computational models.

Proofs of programs are based on "symbolic execution," a generalization of simulation that allows the user to prove the validity of a formula in all temporal models. Proof rules fall into two main types: dynamic and static. The dynamic proof rules cause the execution to advance to future states by direct execution, branching, or looping, for example. The static proof rules are used to prove facts about a given state or set of states. The domains that the theory of a state may concern include many that are useful in computer applications, for example, integers, bitstrings, and arrays. Facts to be proved about a given state are first-order formulas (sometimes involving quantification). The static deduction capability includes, among others, decision procedures where feasible (for example, for boolean ex-

pressions and equality), the simplex algorithm for linear arithmetic, user-invocable axioms for the various domains, and rules for manipulating quantifiers (including internal access to the EKL quantification solver [11]). The various partial decision procedures are linked via the Nelson-Oppen cooperating decision-procedure paradigm [13]. Of course, the user does not explicitly see the internal workings whereby the decision procedures "cooperate," but for the system builder, it makes adding proof capabilities about new domains relatively easy. In addition to the invocable axiom capability, SDVS has a number of system flags that can be set and reset during a proof in order to help regulate the trade-off of system and user effort. For example, certain flags are turned on in cases where the user does not want to give explicit proof commands, but simply to have the system "try harder."

Typically, a proof is developed in an interactive session with SDVS and saved for future reexecution in SDVS or for minor alteration in an editor. In an SDVS session, a proof may be manipulated by "popping" back to a previous proof step, which restores the correct state of affairs at that time. In this way, different proof strategies can be tried in an efficient manner. A batch proof may be created by saving a proof developed interactively or by writing it directly using an editor.

2.2 SDVS Translators

As mentioned above, SDVS is able to handle proofs of correctness for programming and hardware description languages by translating such programs and descriptions into state deltas. The translators are constructed from formal specifications of the languages' semantics in terms of the state delta logic using a tool called DENOTE (Denotational Semantics Translation Environment) [14]. As part of this effort we defined a language called DL (DENOTE Language) for writing equations specifying a language's semantics. DENOTE translates specifications written in DL and outputs either formatted equations or a Common Lisp implementation of the translator. DENOTE was used this year to generate all of our translator implementations.

In addition to increasing our productivity, this method of implementing translators from their formal specifications has led to more reliable, better structured translators. The method also permits us to define incrementally the semantics of an application language to accommodate increasingly larger, more complex subsets of the language. Incremental development also enables us to define and achieve identifiable milestones.

2.3 SDVS User Interface

In addition to the proof commands, there are query commands that can be used to inform the user about various aspects of the current state of the symbolic execution (for example, values of variables or quantified statements considered true) and input-output commands for reading/writing proofs or lemmas from/to files. The user-to-system channel currently is restricted to type-in commands (no icons or menus), and the system-to-user channel consists of English and mathematical notation. However, we are cognizant of the value of a more graphical and flexible interface in making information about the proof more

readily accessible and understandable to the user. Furthermore, to gain acceptance by the targeted user community, it is important to conform to the industry standard "user-friendly" technology.

As an example of a more advanced user interface capability, we have implemented a prototype of windows for tracing the flow of control in an Ada program or VHDL hardware description corresponding to an ongoing symbolic execution proof of correctness for that program. This prototype interface was implemented for another project and supported by the Air Force. It is not included in SDVS 12. However, we plan to add a general X-window interface to SDVS.

2.4 Accomplishments

To date, the main visible accomplishment of the SDVS project has been the creation of a system within which machine-checked verifications of real applications are possible. The largest are the proof of a real microcode verification example written in ISPS (completed in 1986 [15]) and a real Ada program (completed this year [4, 16, 17]). Additionally, we have verified many Ada programs and VHDL descriptions taken from books and libraries with respect to liveness and safety specifications [3, 18–27].

We are currently in the midst of the effort to verify VHDL description of the AMD TAXIchip [28], [29].

3 Software Verification Progress – Ada

3.1 Language Extensions

From 1988 to 1993 we adapted SDVS to handle increasingly larger subsets of Ada. In 1994 we completed work on the MSX verification example; no changes in Ada capability are represented in SDVS 13.

The features of the six Ada subsets that represent stages in the increasing capability of SDVS were as follows:

Core Ada: scalar assignment statements and simple expression evaluation; straight-line program flow; branching (`IF, CASE`), and iteration (`WHILE`) statements; simple input and output (via the `GET` and `PUT` procedures); block structure, scoping and variable declarations; packages with restricted object declarations; `USE` clauses; basic data types (integer, boolean, array).

Stage 1 Ada: the features of Core Ada, plus nonscalar assignment, subprogram declarations and subprogram calls, package bodies, record types, and enumeration types.

Stage 2 Ada: the features of Stage 1 Ada, plus user-defined exception handling and the character data type.

Stage 3 Ada: the features of Stage 2 Ada, plus context clause declarations (for certain I/O subpackages of the STANDARD package), rudimentary overload resolution for subprogram arguments, the string data type, and a preliminary version of floating-point types.

SDVS 11 Ada: the features of Stage 3 Ada, plus `FOR` loops and the elimination of existential quantification of declared variables.

SDVS 12 Ada: the features of SDVS 11 Ada, plus integer subtypes and integer definition types, and for these types: explicit type conversions, length representation clauses (representation clauses specifying an amount of storage associated with a type), and instances of the generic function `UNCHECKED_CONVERSION`; "execution marks" (in the form of interpreted comments) for statements, permitting the user to specify beginning and end points for symbolic execution.

The SDVS 12 additions were required by the Ada application verified last year and completed this year.

The adaptation of SDVS to handle subsets of Ada included writing the formal specifications and implementations of Ada-to-state-delta translators. The items involved in this task included defining the grammar of the Ada subsets, formally specifying the two Ada-to-state-delta translator phases for each subset, building the lexical analyzer and parser for each subset, completing Common Lisp implementations of the two phases of each translator specification, testing the translator implementations, and implementing enhancements to

9

SDVS that allow correctness proofs of Ada programs. We have experimented with the Ada translators and SDVS system modifications by proving the correctness of numerous small Ada programs. This involved constructing the Ada programs and their specifications, as well as the proofs that the programs satisfy the specifications.

More specifically, for each subset we defined both the concrete syntax and the abstract syntax. The parser, using the concrete syntax, generates the abstract syntax for the back end (phases 1 and 2) of the translation. We used the abstract syntax to specify formally the back end of the Ada translator. This specification gives the denotational semantics of the Ada subset in terms of the state delta logic. The entire specification is written in our language DL so that it can be input to DENOTE. The Common Lisp implementation of the translator is automatically generated by DENOTE.

3.2   SDVS/Ada Applications

A "critical mass" of theory and system development had to be established before portions of "real" Ada applications could be verified. As indicated above, the initial Ada capabilities were demonstrated by the verification of Ada programs taken primarily from books and Ada libraries. With a substantial capability implemented, this year's emphasis was on verifying portions of a DoD Ada application. More work is needed to enhance SDVS for sizable applications, and such verification experiments on large programs must be performed to drive further research and development.

At the end of fiscal year 1992, we devised a plan for coordinating efforts at The Aerospace Corporation and the Johns Hopkins University Applied Physics Laboratory (APL) to verify portions of the Midcourse Space Experiment (MSX) software using SDVS. This effort was a "shadow project" (i.e., it did not affect the MSX deliverables schedule), and it was directed toward stressing SDVS and formulating further research strategies.

The software for the MSX Program satisfied many of the requirements that Aerospace had originally defined for an Ada application. The most important reasons for selecting the MSX software were as follows:

- Complete, accurate documentation was available.

- Access to the developers was provided.

- The correct functioning of the software is critical to the mission's success.

- The software is not classified or restricted from access.

- There is the potential of verifying a combination of Ada software and embedded 1750A programs.

Aerospace and APL discussed how portions of the MSX software could be verified, given the current capabilities of SDVS. This work was intended to verify some properties of the programs, while providing data to drive further work on SDVS. The following were the main objectives of the MSX application:

- stress-test SDVS and gain more experience in using SDVS to verify systems;

- identify weak and missing features in the tools and underlying theory;

- prioritize tasks for further research and development, and start to investigate the higher priority tasks;

- provide results of the verification effort to the MSX Program Office;

- provide another example for demonstration and publication;

- evaluate the value added from applying formal methods; and

- acquire more information about what designers need in order to apply formal methods.

In a one-year project we achieved some results for each of these objectives, particularly for the first five objectives. Our main accomplishments include the verification of a substantial, real Ada application program and the enhancements made to SDVS in the course of the project.

The first steps of the project consisted of selecting and analyzing a portion of the MSX Ada code to be verified; examining the documentation pertaining to that portion; and delineating the Ada constructs appearing in that portion but that were not implemented in the SDVS Ada translator at that time. The MSX program had a large amount of code from which to choose our example. We factored out those portions of code that performed significant floating-point calculations because we have only started to study how to reason about floating-point numbers in SDVS [30]. Nevertheless, we still had to deal with Ada software that included tasks, numerous interfaces to 1750A assembly routines, and various Ada constructs that were not then implemented in the SDVS Ada translator. We were able to deal, mostly satisfactorily, with the latter two, but tasks were a major problem; we had to rewrite tasks as procedures and provide an Ada scheduler for them. Because of time constraints on this experiment, the scheduler did not encompass a great number of complicated situations that might arise in the execution of the original software.

The portion of MSX software selected for this experiment consisted of (1) two packages containing three tasks and an interrupt-driven procedure, (2) three packages containing type definitions, (3) one package containing Tartan[1] supplied functions for bit manipulations, and (4) four packages only marginally related to our selected software. Portions of software from these packages constituted our target software and consisted of approximately 900 lines of Ada code.

As part of the verification process, a formal specification of the software was created from the informal documentation supplied by the software developers. Stripped of detail and roughly stated in English, the correctness assertion that we proved for this program is "the Ada program correctly partitions and reformats an error-free infinite input stream of bytes into its constituent data-structure messages."

---

[1]Tartan is the compiler used by APL to compile the MSX software.

In 1993 we attempted two different approaches to the proof: (1) symbolic execution of the Ada portions of the program, including symbolic execution through all invocations of the Ada subprograms and the use of (prototype implementations of) meta-level proof commands (tacticals) to develop subproofs of similar cases; and (2) abstract characterization and proof of properties for major Ada subprograms of the main program, and then the use of these abstract characterizations upon invocation of the subprograms. The first approach was attempted because we thought that the second approach would not be feasible within the time constraints for this experiment. Although the first approach was conceptually easier and the newly implemented meta-proof commands greatly assisted the proof construction, it took too much time to execute the proof. In spite of the long execution time, we completed most of the correctness proof; we proved that 57 of the 61 types of data-structure messages were processed correctly by the software. At the end of 1993, the second approach was attempted, and was not as difficult as first imagined. The characterization and proof for two major subprograms were completed and a third was partially completed. This approach greatly reduces a time/space explosion and will permit the verification of more general schedulers and input.

This project only lasted a year, and because (1) this was the first large Ada application verified and (2) SDVS development occurred concurrently with the program verification, we were restricted in the complexity of the specification and amount of code that could be tackled. However, we found this experience very important for identifying weak and missing features in the tools and underlying theory, and it is helping us to prioritize future research and development tasks. Apart from stress-testing SDVS, the verification project allowed us to enhance the SDVS Ada translator and to prototype three new meta-proof commands to the system that should facilitate lengthy proofs.

Viewed in a larger context, the exercise reported on here, the formal proof of correctness of a part of a real Ada system (with respect to a specification written in the logic of state deltas), is a small preliminary step in the long-term effort of technology transition of formal methods to "industrial" software engineering use, illustrating some of the problems, benefits, and trade-offs of formal verification.

3.3 Technical Reports

The report [4] is the summation paper of the MSX project in publishable format.

4 Hardware Verification Progress – VHDL

Prior to 1987 we adapted SDVS to handle a subset of the hardware description language ISPS. However, ISPS has serious limitations regarding the specification of hardware at levels other than the register transfer level. In 1988 we studied some of the hardware verification research being done outside Aerospace and investigated VHDL, a DoD and IEEE standard hardware description language that was released in December 1987. We selected VHDL as a medium for hardware description within SDVS.

4.1 Language Extensions

From 1989 to 1994 we adapted SDVS to handle increasingly larger subsets of VHDL, in which both combinational and sequential circuits can be described. The features of the five VHDL subsets are as follows:

Core VHDL: ENTITY declarations; ARCHITECTURE bodies; CONSTANT, VARIABLE, SIGNAL, and PORT declarations; BOOLEAN, BIT, INTEGER, and BIT_VECTOR data types; variable and signal assignment statements; IF, CASE, and NULL statements; restricted WAIT statements with inertial delay; and concurrent PROCESS statements.

Stage 1 VHDL: the features of Core VHDL, plus WAIT statements in arbitrary contexts; LOOP, WHILE, and EXIT statements; TRANSPORT delay; aggregate signal assignments; and a revised translator structure.

Stage 2 VHDL: the features of Stage 1 VHDL, plus (restricted) design files, declarative parts in entity declarations, package STANDARD (containing predefined types BOOLEAN, BIT, INTEGER, TIME, CHARACTER, REAL, STRING, and BIT_VECTOR), user-defined packages, USE clauses, array type declarations, enumeration types, subprograms (procedures and functions, excluding parameters of object class SIGNAL), concurrent signal assignment statements, FOR loops, octal and hexadecimal representations of bitstrings, ports of default object class SIGNAL, and general expressions of type TIME in AFTER clauses.

Stage 3 VHDL: the features of Stage 2 VHDL, plus subtypes of scalar types, integer type definitions, and type conversions between integer types; "execution marks" (in the form of interpreted comments) for sequential statements permitting the user to specify beginning and end points for symbolic execution; "VHDL offline characterization" — a facility for specifying, proving, and invoking the behavior of a VHDL subprogram.

Stage 4 VHDL: the features of Stage 3 VHDL, plus design units, context items, component declarations, component instantiation statements, BLOCK statements, generics, generic maps, port maps, and configuration declarations.

Implemented this year, Stage 4 VHDL comprises a significantly more powerful subset of VHDL than the previous stages, in that Stage 4 VHDL admits the *structural description*

of hardware in terms of its hierarchical decomposition into connected subcomponents as outlined in [31]. The earlier VHDL subsets supported only behavioral (e.g., algorithmic or dataflow) hardware descriptions.

The primary VHDL abstraction for modeling a digital device is the *design entity*. A design entity consists of two parts: an *entity declaration* and an *architecture body*. The entity declaration provides the "external view" of the device: it defines the interface between the design entity and its environment, including the number, direction, and type of ports, and corresponds to a *symbol* in a traditional CAE (Computer-Aided Engineering) design methodology. The architecture body provides the "internal view" of the device, describing its behavior or structure, and thereby expressing the relationship between its inputs and outputs. A given entity declaration may be shared by several design entities, each with a different architecture body.

The special case of a structural architecture, in particular, corresponds to the CAE notion of a *schematic*. A structural architecture for a design entity is described by declaring internal signals and connecting these, as well as the ports of the entity declaration, to the ports of various subcomponents declared in *component declarations* and created by *component instantiation statements* in the architecture body.

Component declarations provide a "template" mechanism, whereby an architecture body containing component instantiations can be *analyzed* — checked for syntactic and semantic correctness — independently of prior analysis of entity declarations for those components.

A component instantiation statement specifies an instance of a *child component* occurring inside a *parent component*. At the point of instantiation, only the external view of the child component — the names, types, and directions of its ports — is visible; the child component's internal signals are not visible. The component instantiation statement identifies the child component and specifies which ports or signals in the parent component are connected to which ports in the child component. Component instantiation statements are transformed, in a manner prescribed by the VHDL LRM [32], to pairs of nested BLOCK statements during the elaboration of a VHDL design entity prior to its execution. A BLOCK statement provides a block-structured scope with local declarations and a body consisting of concurrent statements. Elaboration of a design entity recursively transforms component instantiation statements occurring in BLOCK statements until the innermost blocks contain only PROCESS and concurrent signal assignment statements.

The *configuration declaration* provides the mechanism whereby architecture bodies are paired with entity declarations to configure specific design entities. A configuration declaration is analogous to a "parts list," describing which part to use for each component of a hardware design.

Prerequisites to adapting SDVS to handle VHDL are to define VHDL semantics formally in terms of SDVS's underlying logic and to implement a translator from VHDL to the state delta logic. The translator implementation technique for VHDL is analogous to that for Ada (Section 3.1). We have defined both the concrete syntax and the abstract syntax for each VHDL subset, and have implemented a parser for the language. The back end of the translator has two phases. The first phase does static semantic analysis. It performs various

kinds of error checking (e.g. type checking) and collects an environment that associates all names declared in the subject VHDL description with their attributes. The second phase generates the state delta formulas. This phase receives the environment from the first phase, then uses it to translate the VHDL description incrementally into state deltas as the description is symbolically executed in the course of its correctness proof. Most of the back end is specified in a denotational manner; the sequential constructs in our subsets of VHDL are given a formal denotational semantics, while the operational semantics of the concurrent VHDL PROCESS construct is defined in terms of the translator's algorithm for defining the next set of valid state deltas during symbolic execution. The denotational part of the translator specification is written in the language DL so that it can function as input to DENOTE; the part of the translator specified in DL was automatically generated by DENOTE.

4.2 SDVS/VHDL Application

In fiscal year 1993, Aerospace investigated potential hardware verification applications in coordination with NSA/R2. Target VHDL descriptions for an SDVS/VHDL Application were identified at the onset of fiscal year 1994, consisting of VHDL descriptions — developed in-house at NSA — for a set of commercial standard parts, the *Am7968/Am7969 TAXIchip*™ *(Transparent Asynchronous Xmitter-Receiver Interface) Integrated Circuits* designed by Advanced Micro Devices, Inc. (AMD). NSA proposed this VHDL code as a good choice for an SDVS application by virtue of its being representative and relevant: the bit manipulations are typical of those found in digital devices of interest to NSA, and the particular chipset is likely to be retained in the final production unit of the cryptographic device. We want to thank Steve Lobeck of NSA for helping to identify and supply the code.

The TAXIchip Am7968 Transmitter/Am7969 Receiver chipset constitutes a general-purpose interface for high-speed serial communication between two parallel-data hosts, and is used in a prototype cryptographic device currently being built by NSA. In normal operational mode, each Transmitter/Receiver pair is connected over a private *serial link*, which can be a fiber-optic or copper medium. The Transmitter latches a parallel 12-bit message from the sending host on its input pins, encodes it in two stages, serializes it, and shifts the bitstream out to the serial link. The Receiver continuously deserializes the arriving serial bitstream, decodes the resulting parallel bit patterns, and routes the decoded 12-bit message to the receiving host via its output pins. The encoding scheme is based on the ANSI X3T9.5 Committee's *4B/5B* ("4-bit/5-bit") code, in combination with *NRZI (Non-Return-to-Zero-Invert-ones)* encoding.

The SDVS/VHDL Application was directed towards demonstrating the suitability of SDVS to the verification of realistic VHDL hardware descriptions, stress-testing SDVS, and formulating strategies for further research and development in formal verification generally, and particularly in VHDL verification.

To facilitate our work on the SDVS/VHDL Application, we determined that it would be useful to be able to exercise the Application VHDL models with a commercial VHDL simulator. Such simulations would provide important reference points, both for the behavior

of the models and the semantics of the new VHDL constructs being implemented in SDVS (see Section 4.1 above). Consequently, we evaluated two well-regarded tools — the Vantage Spreadsheet and the Model Technology V-System — and concluded that the Vantage simulator would serve our purposes best. This evaluation took into account compatibility issues with the VHDL tool suites being used by NSA. Accordingly, we obtained the Vantage Spreadsheet from NSA as GFE (Government Furnished Equipment).

In order to better understand the problems involved with specifying and proving properties of the TAXIchip VHDL descriptions, we first created simplified versions of those descriptions. We then wrote specifications for these descriptions, and a specification for a system in which the output of the transmitter was input to the receiver (the VHDL description of this system, as well as those of the transmitter and receiver, employ the newly added SDVS features allowing structural descriptions).

In fiscal year 1995, we intend to proceed by incrementally incorporating additional features of the original TAXIchip descriptions into the simplified descriptions we have produced, and by attempting to prove successively more interesting properties of the latter. Regarding the TAXIchip VHDL as originally given, our goal in principle is to prove (at least) suitable versions of the specifications presented in Sections 4.2.1, 4.2.2, and 4.2.3. This will entail a few, relatively minor, enhancements to the VHDL translator, principally the implementation of a subset of the IEEE STD_LOGIC_1164 multivalue logic system.

4.2.1   Specification for the Combined System

So far, we have done the most work with a structural VHDL description which combines the simplified transmitter and receiver into a single system. Included in this description is an internal driver connected to the X1 lines of both chips to run the internal clocks on both chips. This system accepts four bits in parallel on its input lines, which are sent to the transmitter. The transmitter's output is input via a serial line to the receiver, where the NRZI encoded bits on the serial line are decoded back into the original input, which is then output on the system's output lines. We will go into more detail on how the simplified transmitter and receiver work in the sections devoted to them below.

The state delta *system.sd* below serves as a specification for the system. It states that if the input to the system (represented by variable *system_in*) is acceptable (i.e., it consists of four bits), it will eventually appear as output. The formula *input_ok* states that the input is acceptable, and the formula *input_equals_output* states that the eventual value of the variable *system_out* (which represents the output of the system) equals the original value of *system_in*.

```
[sd pre:  vhdl(system)
    mod:  all
   post:  vhdl_model_elaboration_complete(system),
          [sd pre:  formula(input_ok)
            comod:  all
              mod:  all
```

```
                    post: formula(input_equals_output)]]
```

(The above method for writing such a specification is dictated by the fact that variables do not exist until they are elaborated, so we have to wait until a point when *system_in* exists before stating that the input it represents is acceptable.)

The proof of *system.sd* is straightforward. After using the command **go** to reach a point at which all of the variables are elaborated, a proof of the nested state delta in the postcondition is opened. A case split is done on each of the possible values of the input, and in each case we again use the command **go** to reach a state where the output equals the input. Actually, from the VHDL description of the simplified transmitter, we know that the output becomes equal to the input exactly 30.5 nanoseconds after the system starts executing. The above state delta can be modified to reflect this, and the same proof should work for the modified state delta.

In the future, we hope to examine a more general specification: rather than simply stating that the inputs at the time the execution starts are eventually output, we would like to state that (under certain stability assumptions on the input) the inputs at *any time* will eventually appear on the output. This is a safety property, and although we have a facility for proving safety properties which has been successfully used on Ada programs [33], we have not had occasion to use this facility on VHDL descriptions.

4.2.2   Specification for the Transmitter

We first give a high-level functional description of the simplified transmitter. It is driven by an external line X1. An external driver on this line serves as input to a internal clock which produces two internal clock signals, one slow and one fast. On the edge of each slow clock cycle, the transmitter accepts four bits in parallel, and outputs the five bit NRZI-encoded form of the input in order, most significant digit first, on a serial line on the rising edge of each fast clock cycle. The main differences between the original transmitter and the simplified version are that the control part of the transmitter has been simplified, there is no buffering of inputs inside the transmitter, and there are no SYNC bytes generated.

We now examine how the components of the transmitter interact. On each edge of a slow clock cycle, the encoder component accepts four parallel bits of input, and puts the NRZI encoding onto a five bit wide bus connected to the shifter component. A one-shot control circuit enables the shifter to latch in the values from the bus on the first rising edge of the fast clock after each slow clock cycle. The shifter shifts out the values from the bus in order onto a serial line on each rising fast clock edge.

It seems clear that a specification of this description should express that each digit of the NRZI encoding of the input appears in order on the serial line at some later time. Due to the nature of the simplified description, we actually know the times at which these values are first asserted on the line and how long they are asserted, so we can use the SDVS variable *vhdltime* (which represents the time elapsed since the beginning of the execution of the VHDL description) in the specification. The usual way of describing a sequence of events

in time in the state delta language is to use nested state deltas. By a nested state delta, we mean a state delta formula that has another state delta formula in its postcondition (the state delta *system.sd* earlier in this section is such a state delta).

Having made these remarks, we now give the state delta specification for the transmitter, *transmitter.sd*:

```
[sd pre: vhdl(transmitter)
   comod: all
     mod: all
    post: vhdl_model_elaboration_complete(transmitter),
          formula(nrzi.sd)]
```

where *nrzi.sd* is a nested state delta describing the sequence of values which appear on the output.

We have begun the proof of this state delta and expect to have it finished in the near future. As mentioned earlier, we will also attempt enhance the simplified transmitter by incrementally adding features of the original transmitter and attempting to prove more interesting properties.

4.2.3   Specification for the Receiver

As in the preceding section, we begin with a high-level functional description of the simplified receiver. Like the transmitter, it is driven by an external line X1, and an external driver on this line serves as input to a internal clock which produces two internal clock signals, one slow and one fast. On each falling fast clock edge, the receiver accepts input on its input serial line. After accepting five such inputs, the receiver outputs the NRZI decoded form at the end of a slow clock cycle. The differences between the simplified description of the receiver and the original are similar to the differences between the simplified transmitter and the original.

The components of the receiver interact as follows. The shift_and_count component shifts in input from the serial input line, outputs the shifted form onto a bus, and keeps count of how many bits of the NRZI encoding have been accepted. When five bits have been accepted, the shift_and_count component enables the decode component to latch in the values on the bus, and then decode them and output the decoded version.

Since the receiver is intended to work in conjunction with the transmitter, the informal specification is that the output of the receiver must be the NRZI decoding of the values which appear at the on the serial input line at first five falling edges of the fast clock. Again, as with the transmitter, we know the times at which these falling edges occur, so we can use the variable *vhdltime* in our specification.

The state delta specification for the receiver, *receiver.sd* is given below:

```
[sd pre: vhdl(receiver),
```

```
              input_sequence(b1,b2,b3,b4,b5),
              nrzi_ok(b1,b2,b3,b4,b5)
      comod: all
        mod: all
       post: #receiver_out = nrzi_decoding(b1,b2,b3,b4,b5)]
```

The variable *receiver_out* corresponds to the four-bit parallel output lines. The macro *input_sequence* states that the sequence of bits accepted by the receiver on the first five falling edges of the fast clock are b1, b2, b3, b4, and b5. The macro *nrzi_ok* states that these five bits are a legitimate nrzi encoding. The macro *nrzi_decoding* decodes the bit sequence b1,b2,b3,b4,b5. Thus the above state delta states that if the receiver receives on its serial input line a legitimate NRZI encoded byte, it eventually outputs the decoded version of that byte.

As mentioned earlier, using a state delta which characterizes input (such as *input_sequence*) in a proof presents problems, and thus proving a state delta such as *receiver.sd* is not straightforward. Nevertheless, we have mapped out a plan that, if successful, will allow us to demonstrate that the description of the receiver meets its specification. We intend to carry out this plan in the near future.

4.3   Technical Reports

Reference [29] specifies our plan for accomplishing the SDVS/VHDL Application task, including background, objectives, and expected results.

Reference [34] documents a formal semantic specification of Stage 4 VHDL. Now implemented in SDVS, the Stage 4 VHDL translator represents the latest phase of our research on proving properties of VHDL descriptions. The semantics is primarily specified denotationally, although the second-phase semantics of the VHDL simulation cycle has a direct operational implementation in the VHDL translator code.

In addition, we made a presentation entitled "VHDL Verification Using SDVS" at the Fourth Space INFOSEC Symposium, jointly sponsored by the National Security Agency and Aerospace and held April 5-7 at The Aerospace Corporation. After providing a general introduction to formal methods and computer verification, we described the components of SDVS, the underlying model of computation, the proof procedure, the current VHDL capability, and completed and anticipated applications.

5   General System Development

5.1   Development

In 1994 we completed the latest version of SDVS, referred to as SDVS 13, along with a description of the installation procedure and an enlarged (and reorganized) suite of proofs used for validating SDVS installations.

SDVS 13 currently runs under Franz Allegro Common Lisp (FACL) 4.2 on either Sparc 2 or Sparc 10 processors. Aerospace has a "runtime generator" license agreement from Franz, Inc., which allows Aerospace to deliver SDVS to end users without requiring them to purchase a Common Lisp system (the runtime version removes some features from the development version of Common Lisp).

The *SDVS 13 Users' Manual* [35] has been updated to reflect the new subsets of VHDL, and all examples in the manual have been executed using SDVS 13.

SDVS has been approved for the following organizations: Cambridge University, George Mason University, Johns Hopkins University Applied Physics Laboratory, MITRE Corporation, National Security Agency (NSA), Naval Research Laboratory, Rome Laboratory, Trusted Information Systems, and University of California at Santa Barbara.

Last year Aerospace developed a (short) SDVS Software Request Form in order to simplify and expedite the approval process; online versions can now be sent via e-mail (sdvs-requests@aero.org). We continued to distribute technical reports and publications to those individuals who have expressed an interest in this work.

In 1994, SDVS was presented at several fora, including

1. the NSA,

2. the Launch Range Management Conference IV,

3. the 1994 Space INFOSEC Symposium and

4. the Aerospace Ada Interest Group.

A facility is available that permits "remote demonstrations" of SDVS. This facility allows Aerospace to give a demonstration of the capabilities of SDVS to users running the X-windows system at any site on the Internet. When the remote demonstration facility is invoked, a window appears on a remote display containing a trace of the local activity of an SDVS session. Control of the SDVS session remains solely with the local Aerospace user; i.e., the remote user has no input capability to the SDVS session.

Under an Air Force sponsored project, we are continuing to explore the potential for expanding the SDVS user interface to make use of bitmapped displays and windowing technologies. Specifically, we are using X-windows as our windowing platform. X-windows is quickly becoming a portable standard windowing system. Previously, we started to build the interface using Common Windows, but found this effort too time-consuming and difficult.

5.2 Technical Reports

Reference [35] is the users' manual for SDVS 13. Although it is primarily a reference manual, it has tutorial aspects as well. The manual contains descriptions of the following:

- underlying logic (state delta logic)
- proof language
- user interface
- ISPS verification capability
- Ada verification capability
- VHDL verification capability
- domains defined in the SDVS Simplifier and capabilities of static solvers

All facets of the system are illustrated with example SDVS sessions.

Reference [36] is a tutorial for SDVS, containing numerous examples of proofs in SDVS. In particular, it contains descriptions and examples of the following:

- state delta logic
- dynamic and static proof commands
- some SDVS data types
- quantification
- techniques for verifying hardware descriptions and programs written in VHDL, Ada, or ISPS

6 Formal Specification Languages

The early phase of this year's contract provided for an examination of standard formal specification languages. In this effort we conducted a preliminary examination of the languages Z [37], VDM [38], LOTOS [39], Larch [40], and Estelle [41]. The goal was to gain some familiarity with the languages, their uses in applications, and their potential for incorporation into a formal verification methodology. Unfortunately, this effort was cut off before any specific conclusions could be drawn or any report written.

However, we did continue related work under an Aerospace Sponsored Research program. This work primarily focussed on Z and its tools FuZZ and ProofPower.

7 Conclusion

The progress in FY 94 was primarily in the SDVS capability to handle VHDL descriptions, both in the translation and proof areas.

The translator implementation technique that we developed in 1987 has continued to be very successful, and it has been used this year for the translator implementations of VHDL.

This year's VHDL work, with its projected completion next year, has added the third leg, the hardware leg, in the triad of real applications of SDVS: to software, firmware, and hardware verification.

In addition we wrapped up the MSX Ada verification example, and made a preliminary study of formal specification languages.

Next year the emphasis will be on applying formal specification techniques to the Synergy project [42]. In addition we will complete the TAXIchip VHDL verification example, and will continue to push and work for technology transfer of formal specification and verification methods to real users.

References

[1] J. V. Cook, I. V. Filippenko, B. H. Levy, L. G. Marcus, and T. K. Menas, "Formal Computer Verification in the State Delta Verification System (SDVS)," in *Proceedings of the AIAA Computing in Aerospace Conference*, (Baltimore, Maryland), pp. 77–87, American Institute of Aeronautics and Astronautics, October 1991.

[2] B. H. Levy, "An Overview of Hardware Verification Using the State Delta Verification System (SDVS)," in *Proceedings of the 1991 International Workshop on Formal Methods in VLSI Design*, (Miami, Fla.), ACM, January 1991.

[3] B. Levy, I. Filippenko, L. Marcus, and T. Menas, "Using the State Delta Verification System (SDVS) for Hardware Verification," in *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience: Nijmegen, The Netherlands* (ed. V. Stavridou, T. F. Melham, and R. T. Boute), pp. 337–360, North-Holland, June 1992.

[4] T. K. Menas, J. M. Bouler, J. E. Doner, I. V. Filippenko, B. H. Levy, and L. G. Marcus, "Using SDVS to Assess the Correctness of Ada Software Used in the Midcourse Space Experiment," Technical Report ATR-94(4778)-1, The Aerospace Corporation, April 1994.

[5] J. V. Cook, "Final Report for the C/30 Microcode Verification Project," Technical Report ATR-86(6771)-3, The Aerospace Corporation, September 1986.

[6] I. V. Filippenko, "VHDL Verification in the State Delta Verification System (SDVS)," in *Proceedings of the 1991 International Workshop on Formal Methods in VLSI Design* (ed. P. A. Subrahmanyam), (Miami, Fla.), ACM, January 1991.

[7] L. G. Marcus, "The Incorporation of Testing into Verification: Direct, Modular, and Hierarchical Correctness Degrees," in *Proceedings of the 1994 International Symposium on Software Testing and Analysis*, (Seattle, WA), p. 197, ACM, August 1994.

[8] J. Bowen and M. Gordon, "Z and HOL," in *Z User Workshop, Cambridge 1994* (ed. J. Bowen and J. Hall), Workshops in Computing, pp. 141–167, Springer-Verlag, 1994.

[9] L. Marcus, T. Redmond, and S. Shelah, "Completeness of State Deltas," Technical Report ATR-86(8454)-2, The Aerospace Corporation, September 1986.

[10] T. K. Menas, "The Relation of the Temporal Logic of the State Delta Verification System (SDVS) to Classical First-Order Temporal Logic," Technical Report ATR-90(5778)-10, The Aerospace Corporation, September 1990.

[11] J. Ketonen and J. Weening, "EKL—An Interactive Proof Checker User's Reference Manual," Technical Report STAN-CS-84-1006, Dept. of Computer Science, Stanford University, June 1984.

[12] *Penelope Volume 2: Penelope Ada Verification System User Manuals*, (Ithaca, NY: Odyssey Research Associates, Inc., 1990).

[13] G. Nelson and D. C. Oppen, "Simplification by Cooperating Decision Procedures," *ACM Trans. Programming Languages and Systems*, Vol. 1, pp. 245–257, October 1979.

[14] J. V. Cook, "The Language for DENOTE (Denotational Semantics Translation Environment)," Technical Report TR-0090(5920-07)-2, The Aerospace Corporation, September 1990.

[15] J. V. Cook, "Verification of the C/30 Microcode Using the State Delta Verification System (SDVS)," in *Proceedings of the 13th National Computer Security Conference*, (Washington, D. C.), pp. 20–31, National Institute of Standards and Technology/National Computer Security Center, October 1990.

[16] T. K. Menas, J. M. Bouler, and J. E. Doner, "Specifications and Correctness Proofs for Portions of the MSX Ada Software," Technical Report ATR-93(3778)-5, The Aerospace Corporation, September 1993.

[17] T. K. Menas, J. M. Bouler, J. E. Doner, I. V. Filippenko, B. H. Levy, and L. G. Marcus, "Overview of the MSX Verification Experiment using SDVS," Technical Report ATR-93(3778)-6, The Aerospace Corporation, September 1993.

[18] I. V. Filippenko, "Example Proof of a Core VHDL Description in the State Delta Verification System (SDVS)," Technical Report ATR-90(5778)-6, The Aerospace Corporation, September 1990.

[19] I. V. Filippenko, "Some Examples of Verifying Core VHDL Hardware Descriptions Using the State Delta Verification System (SDVS)," Technical Report ATR-91(6778)-6, The Aerospace Corporation, September 1991.

[20] I. V. Filippenko, J. M. Bouler, and B. H. Levy, "Some Examples of Verifying Stage 1 VHDL Hardware Descriptions Using the State Delta Verification System (SDVS)," Technical Report ATR-92(2778)-3, The Aerospace Corporation, September 1992.

[21] I. V. Filippenko, "Some Examples of Verifying Stage 3 VHDL Hardware Descriptions Using SDVS," Technical Report ATR-93(3778)-1, The Aerospace Corporation, September 1993.

[22] J. V. Cook and J. E. Doner, "A Modular Correctness Proof of a Quicksort Procedure Written in Ada using SDVS," Technical Report ATR-91(6778)-8, The Aerospace Corporation, September 1991.

[23] T. A. Aiken, J. V. Cook, and L. G. Marcus, "Example Proofs of Core Ada Programs in the State Delta Verification System," Technical Report ATR-88(3778)-6, The Aerospace Corporation, September 1988.

[24] J. Doner and J. V. Cook, "Example Proofs of Stage 1 Ada Programs in the State Delta Verification System (SDVS)," Technical Report ATR-89(4778)-8, The Aerospace Corporation, September 1989.

[25] J. V. Cook and D. F. Martin, "Example Proofs of Stage 2 Ada Programs Containing Exceptions in the State Delta Verification System (SDVS)," Technical Report ATR-90(5778)-11, The Aerospace Corporation, September 1990.

[26] J. V. Cook, "Example Proofs of Stage 3 Ada Programs in the State Delta Verification System (SDVS)," Technical Report ATR-91(6778)-3, The Aerospace Corporation, September 1991.

[27] J. V. Cook and J. E. Doner, "Example Proofs Using Offline Characterization of Procedures in the State Delta Verification System (SDVS)," Technical Report TR-0090(5920-07)-3, The Aerospace Corporation, September 1990.

[28] Advanced Micro Devices, Inc., *Am7968/Am7969 TAXIchip Integrated Circuits Technical Manual*, October 1992.

[29] I. V. Filippenko, "SDVS/VHDL Application Program Plan," Technical Report ATR-94(4778)-2, The Aerospace Corporation, March 1994.

[30] L. G. Marcus, "Preliminary Investigations into Specifying and Proving Ada Floating-Point Programs in the State Delta Verification System (SDVS)," Technical Report ATR-91(6778)-4, The Aerospace Corporation, September 1991.

[31] I. V. Filippenko and L. G. Marcus, "Integrating Structural VHDL Hardware Descriptions into the State Delta Verification System (SDVS)," Technical Report ATR-92(8180)-1, The Aerospace Corporation, September 1992.

[32] IEEE, *Standard VHDL Language Reference Manual*, 1988. IEEE Std. 1076-1987.

[33] T. Menas, "Safety Properties of Terminating and Nonterminating Ada Programs in SDVS," Technical Report ATR-92(2778)-2, The Aerospace Corporation, September 1992.

[34] I. V. Filippenko, "A Formal Description of the Incremental Translation of Stage 4 VHDL into State Deltas in SDVS," Technical Report ATR-94(4778)-4, The Aerospace Corporation, September 1994.

[35] L. G. Marcus, "SDVS 13 Users' Manual," Technical Report ATR-94(4778)-5, The Aerospace Corporation, September 1994.

[36] T. K. Menas and I. V. Filippenko, "SDVS 13 Tutorial," Technical Report ATR-94(4778)-6, The Aerospace Corporation, September 1994.

[37] J. M. Spivey, *The Z Notation*, (Hertfordshire, U. K.: Prentice Hall, 1989).

[38] British Standards Institute, *VDM Specification Language: Proto-Standard (IST/5/50)*, 1989.

[39] International Organization for Standardization, Geneva, *Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique based on the Temporal Ordering of Observational Behavior (ISO/IEC 8807)*, 1991.

[40] J. V. Guttag, J. J. Horning, and J. M. Wing, "The Larch Family of specification languages," *IEEE Software*, Vol. 2, pp. 24–36, September 1986.

[41] International Organization for Standardization, Geneva, *Information Processing Systems – Open Systems Interconnection – ESTELLE – A Formal Description Technique based on an Extended State Transition Model (ISO/IEC 9074)*, 1991.

[42] O. S. Saydjari, S. J. Turner, D. E. Peele, J. F. Farrell, P. A. Loscocco, W. Kutz, and G. L. Bock, "Synergy: A Distributed Microkernel-based Security Architecture." National Security Agency INFOSEC Research and Technology, 22 November 1993.